Delay and FIR (Matlab Part)

Real-Time Block Simulation in MATLAB

Objective

In this assignment, you will

- Learn about buffer-oriented signal processing
- Use MATLAB to prototype and debug real-time DSP blocks
- Study the operation of a simple delay block
- Design and test an FIR filter block in MATLAB

Pre-lab

Please carefully read the background section and be prepared to answer questions on it.

Background

This section gives some important background material that you should read before coming to the lab.

Real-Time DSP Applications

There are two characteristics of a typical MATLAB implementation of a signal processing algorithm:

1. All of the input samples are available at once.
2. The exact time for the algorithm to run is not critical.

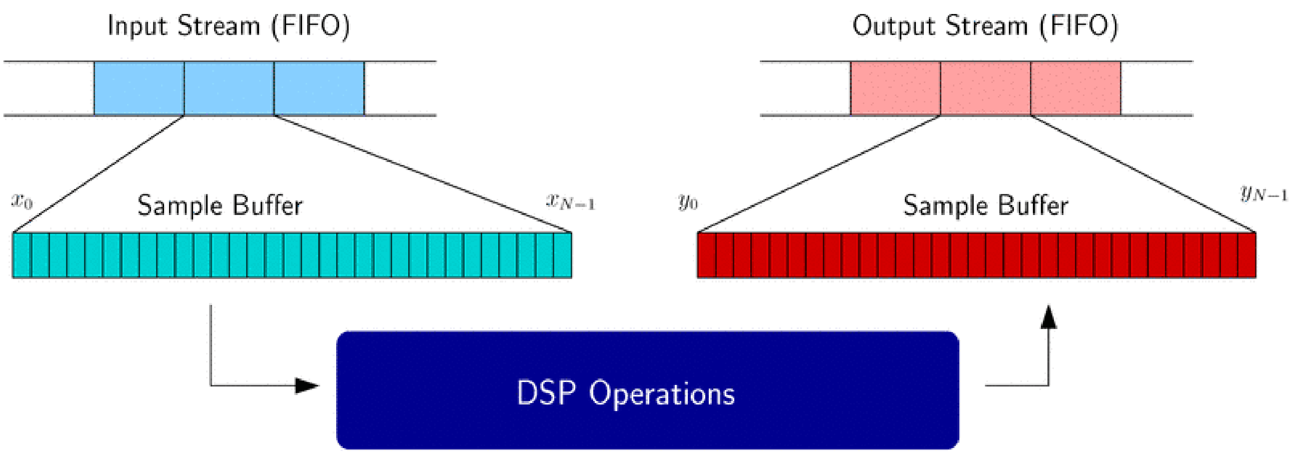
An example non-real-time MATLAB program would be an application that enhances some feature in a digitized image and stores the result as a JPEG file.

Real-time applications differ from a standard MATLAB program:

1. A continuous stream of signals or data is received, which must also be processed continuously.
2. Timing is critical, and the algorithm must not introduce too much *latency*, defined loosely as the input to output delay of the system.

An example real-time signal processing application is echo cancellation in a telephone network. Audio signals must be processed in small enough chunks so that there is no noticeable delay to the speakers. Control applications, where the system must take sensor data and provide proper stimulus in real time are also very sensitive to latency.

Real-time processing in DSPs is usually implemented using a buffer-based strategy as depicted below:



Since we cannot wait for all samples to be ready before we start processing, the input sample stream is divided into buffers of *N* samples. The DSP algorithm then operates on one buffer (or chunk of samples) at a time and processes the *N* input samples to generate *N* samples in the output stream. Note that in general the number of input and output samples would not have to be equal, since we may reduce the number of samples required to represent the important information.

One problem with buffer-based arrangement is that the buffer boundaries are artificial, and we may need some information from previous buffers in order to process the current buffer. In order to do this, we must often store some *state* when we have completed processing the samples for the current buffer, allowing later buffers to be processed correctly.

Note that the size *N* of the buffers is generally a tradeoff between the latency and the throughput. A small buffer size means that the latency will be small, since we do not have to wait for a large buffer to fill up. But, processing many small chunks means that we will waste time storing and restoring state, which means overall lower throughput. The application constraints will typically dictate the optimal value for *N*.

Real-Time DSP Simulation in MATLAB

Although real-time DSP applications are usually coded in C or assembly language, MATLAB can serve as an invaluable first step for prototyping and testing an algorithm before coding it for the DSP. Many simple errors and problems can be flushed out using the user-friendly MATLAB environment in much less time than would be required on the DSP. Especially for this lab, you will save time and gain valuable insight by studying your algorithms in MATLAB before you code them in C.

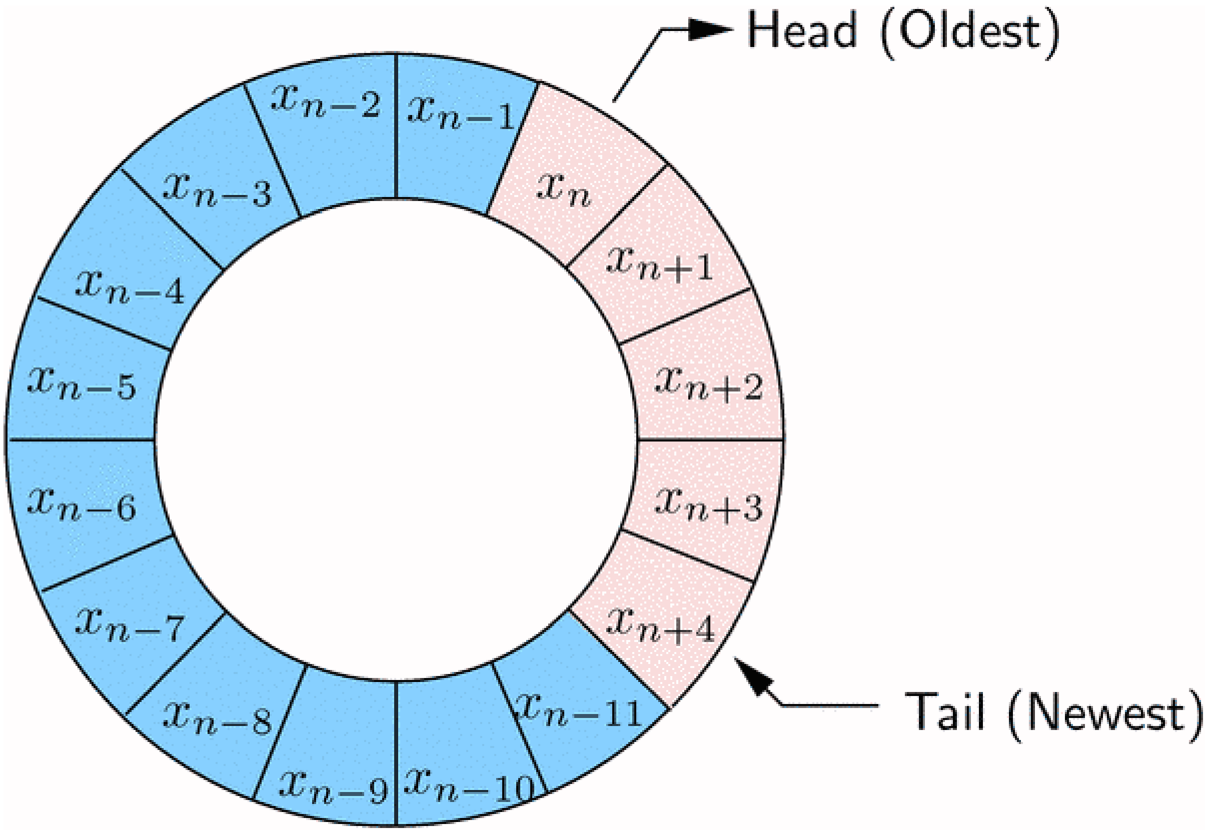
DSP algorithms can usually be described using a block diagram as we did in the Simulink labs. This object-oriented approach not only allows the operations to be coded in an independent fashion, but also the processing blocks to be reused and connected in arbitrary ways. In the following labs, we will take the approach of describing the various DSP operations with logical blocks that then can be connected to perform more complex tasks.

In order to simulate a real-time block in MATLAB, you will learn in this lab how to define MATLAB functions that can

- Operate on a stream of samples in a buffer-oriented fashion
- Pass state from one call to the next

Circular Buffers

One way to simulate a continuous stream of samples with finite-sized buffers is to use circular buffers. The following figure depicts a conceptual representation of a circular buffer.

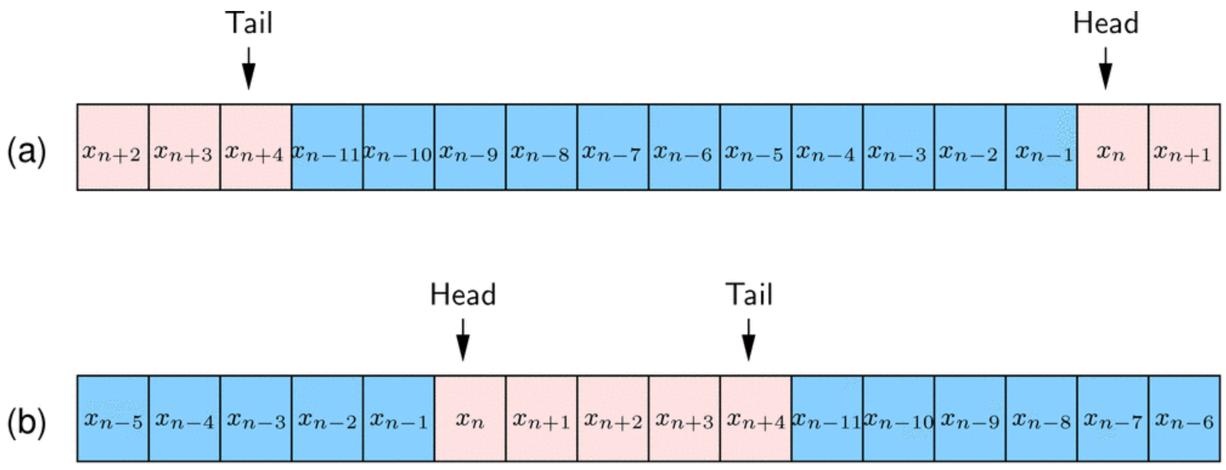


Memory elements are arranged in a circle, where new incoming data items (samples) are placed at the *tail* of the buffer, and the tail pointer is then incremented. During processing, the oldest item not yet processed is taken from the *head* of the buffer, after which the head pointer is also incremented. Note that a circular buffer is also referred to as a first-in first-out (FIFO) buffer as well.

You can think of a circular buffer like a queue or checkout line at the supermarket, where instead of having the people move, both the head (checkout) and the tail (end of the line) move. The reason this arrangement is good for DSP is because we can avoid having to copy or move data. Instead, we only need to update pointers, which is usually much more efficient.

Think of an assembly line of very heavy machines, where it might be more efficient for the workers to move from one machine to the next, rather than to move the machines at each step.

In a processor, we don’t actually store data in a circle, but rather data is arranged logically in a linear fashion, so the circular buffer is “flattened” as shown below.



Tail
Head
(a)
x_{n+2} x_{n+3} x_{n+4} x_{n-11} x_{n-10} x_{n-9} x_{n-8} x_{n-7} x_{n-6} x_{n-5} x_{n-4} x_{n-3} x_{n-2} x_{n-1} x_n x_{n+1}
Head
Tail
(b)
x_{n-5} x_{n-4} x_{n-3} x_{n-2} x_{n-1} x_n x_{n+1} x_{n+2} x_{n+3} x_{n+4} x_{n-11} x_{n-10} x_{n-9} x_{n-8} x_{n-7} x_{n-6}

The only difficulty of this arrangement is that the pointers we use to access the buffer need to somehow wrap from the end back to the beginning of the buffer to simulate the operation of the circular buffer. This is illustrated by showing two possible cases, one where the currently used samples actually wrap across the buffer boundary. Many DSPs have built-in support for circular (wrap-around) addressing, but this usually requires some assembly language programming, which is beyond the scope of this lab.

In this lab, we will simulate a circular buffer in MATLAB and C by making our buffer size an integer power of 2. A simple bit masking operation can then be used to make array indices circular. Although not the most efficient way, you will learn the basic concept of circular addressing, and later you can learn about the detailed assembly instructions needed to automatically support this without too much difficulty.

To illustrate how circular buffering can be accomplished with simple bit masking, consider the following conceptual code:

```
int samples[16];

unsigned int head=0;

unsigned int tail=0;



void put_sample(int samp1)

{

  /* Put a sample at the tail of the buffer */

  samples[tail] = samp1;
  /* Increment.        By masking bits, make modulo 16 */

  tail = (tail + 1) & 15;

}



int get_sample()

{

    int samp1;



    /* Get a sample from head of buffer. */

    samp1 = samples[head];



    /* Increment to next value */

    head = (head + 1) & 15;

}
```

As you can see, we have defined a FIFO or circular buffer with 16 samples, and head and tail indices. The put_sample() function places a single new sample at the tail of the buffer and the get_sample() function reads a sample from the head of the buffer. The important thing to note is the incrementing operations. Instead of just adding one, we AND with *N*-1, where *N* is the size of the buffer. This has the effect of making the additions modulo-N, which is what we need to ensure that we wrap from the end to the beginning of the buffer. Note that decrementing and AND’ing with *N*-1 also works as long as head and tail are unsigned integers.

FIR Filter Design (Window Method)

Perhaps the most common task in digital signal processing is implementing a filter with some given response. A causal linear filtering operation in the time domain can be written as

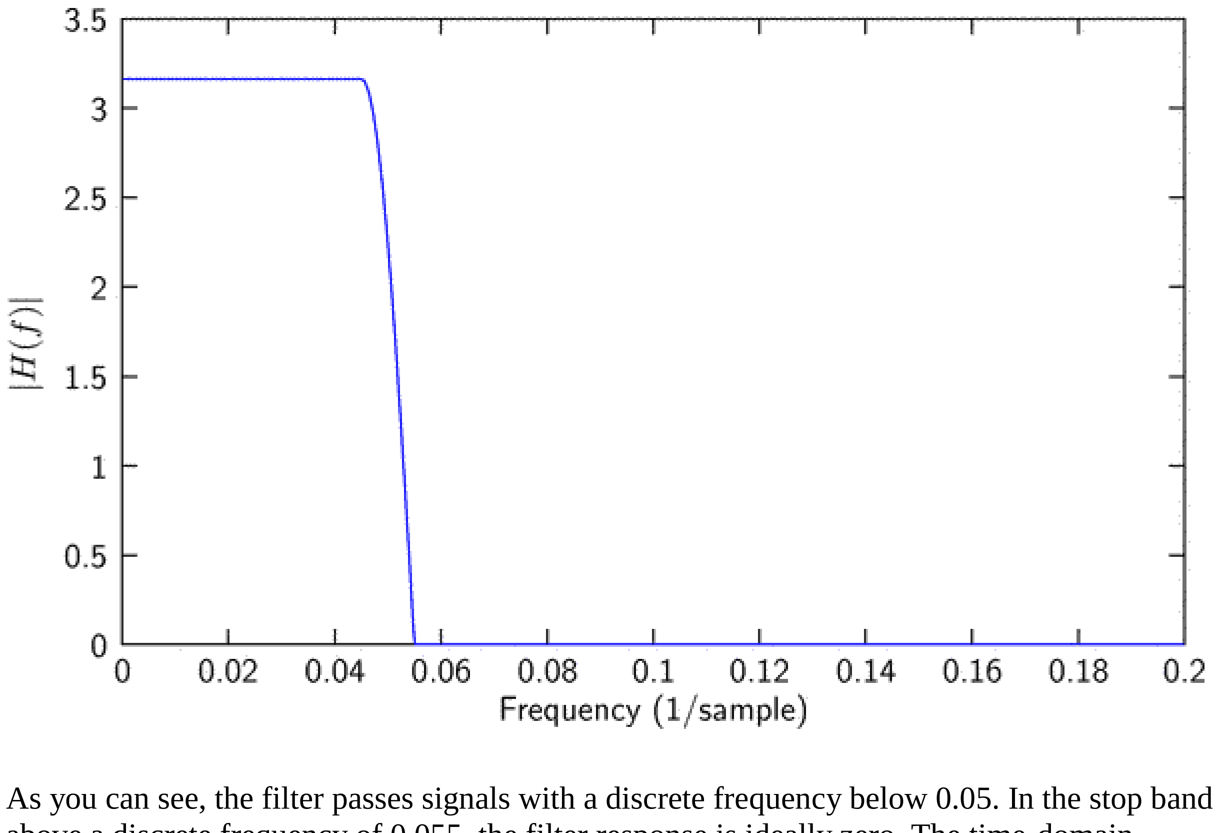
$$y[n] = \sum_{k=1}^{N} a_k y[n-k] + \sum_{k=0}^{M} b_k x[n-k]$$

In this general case, the output y[n] depends on both the input x[n], as well as previous values of y[n]. There are many different ways of specifying the coefficients *a* and *b* depending on the desired filter response and different approximations for mapping continuous time or frequency to discrete time.

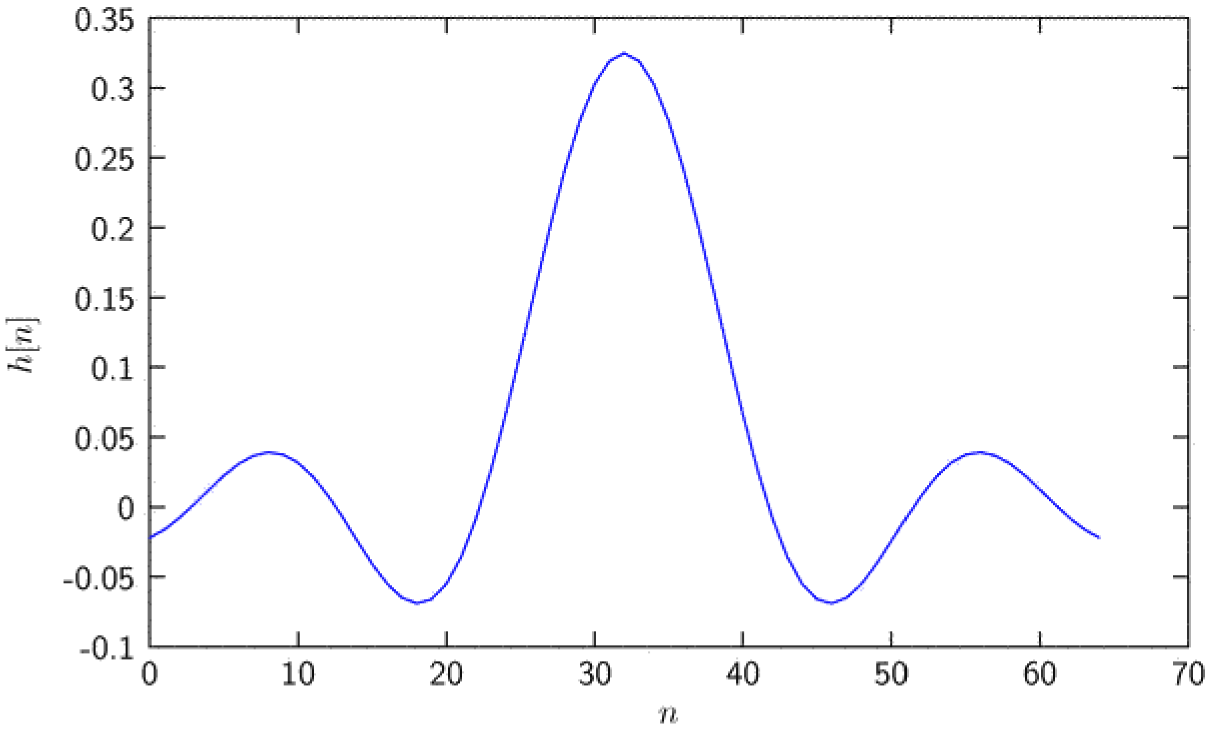
In this assignment, we will consider just one simple filter design technique for finite impulse response (FIR) filters. For an FIR filter, all of the *a* coefficients are equal to 0. In the window method, the filter response is specified in the frequency domain and transformed back to the time domain with an inverse discrete-time Fourier transform (IDTFT). For simple filter types, the inverse Fourier transform can be accomplished in closed form. In the general case, you can compute the IDTFT numerically.

For DSP programming, we would like to make the response of the filter as short as possible in the time domain, and we accomplish this by multiplying the IDTFT of the signal by a window function w[n] of length *M*. There is a fundamental tradeoff between the efficiency and accuracy of our filter, however. If we make *M* small, the DSP code will run faster, but unfortunately the deviation from the specified filter response will be higher; the frequency domain response is "smeared out". In a real application, the specification will determine the best tradeoff between accuracy and efficiency.

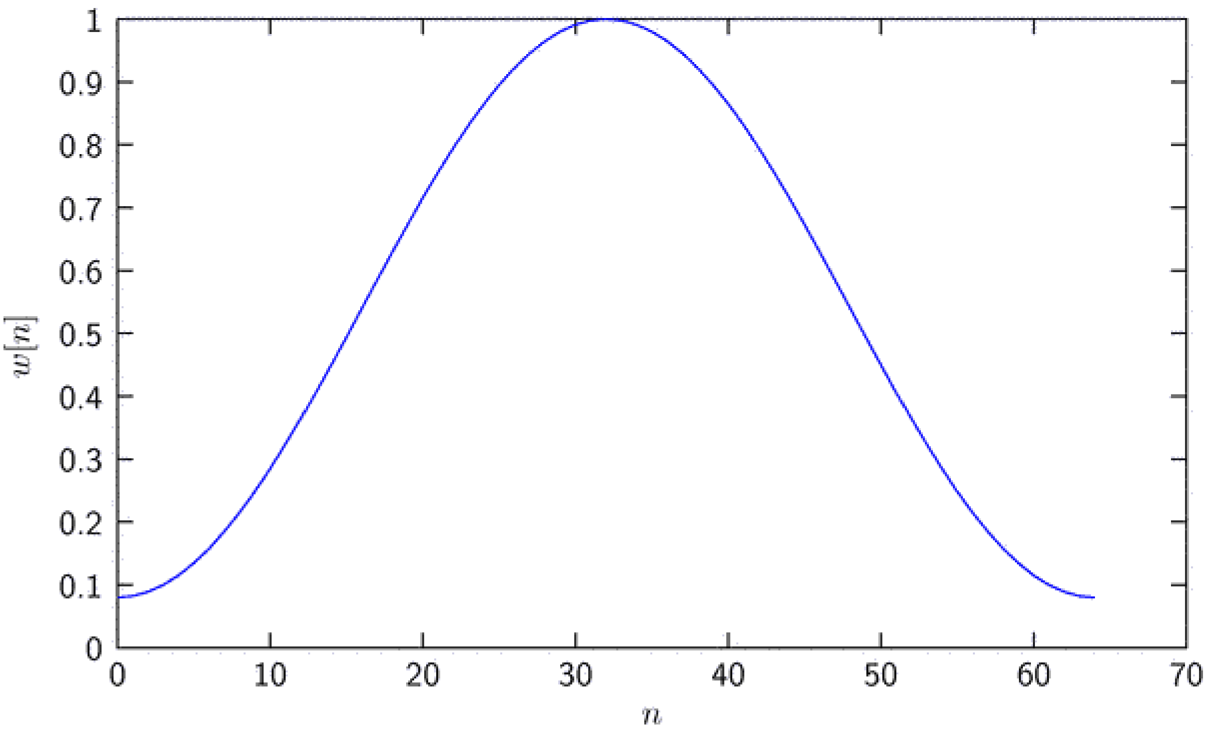
A root-raised cosine filter with a symbol rate of 0.1 and rolloff beta=0.1 is depicted below:



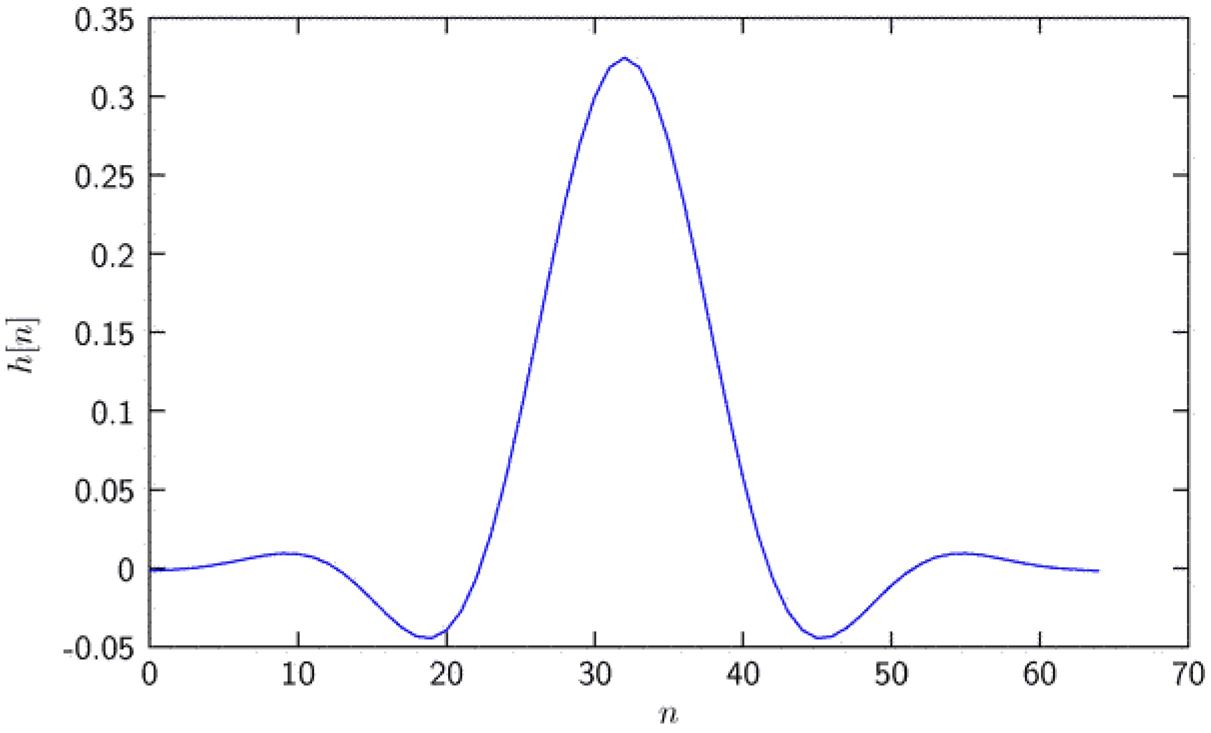
As you can see, the filter passes signals with a discrete frequency below 0.05. In the stop band above a discrete frequency of 0.055, the filter response is ideally zero. The time-domain response of the filter is found by numerically computing the IDTFT. This time-domain response for *M*=64 taps is shown below:



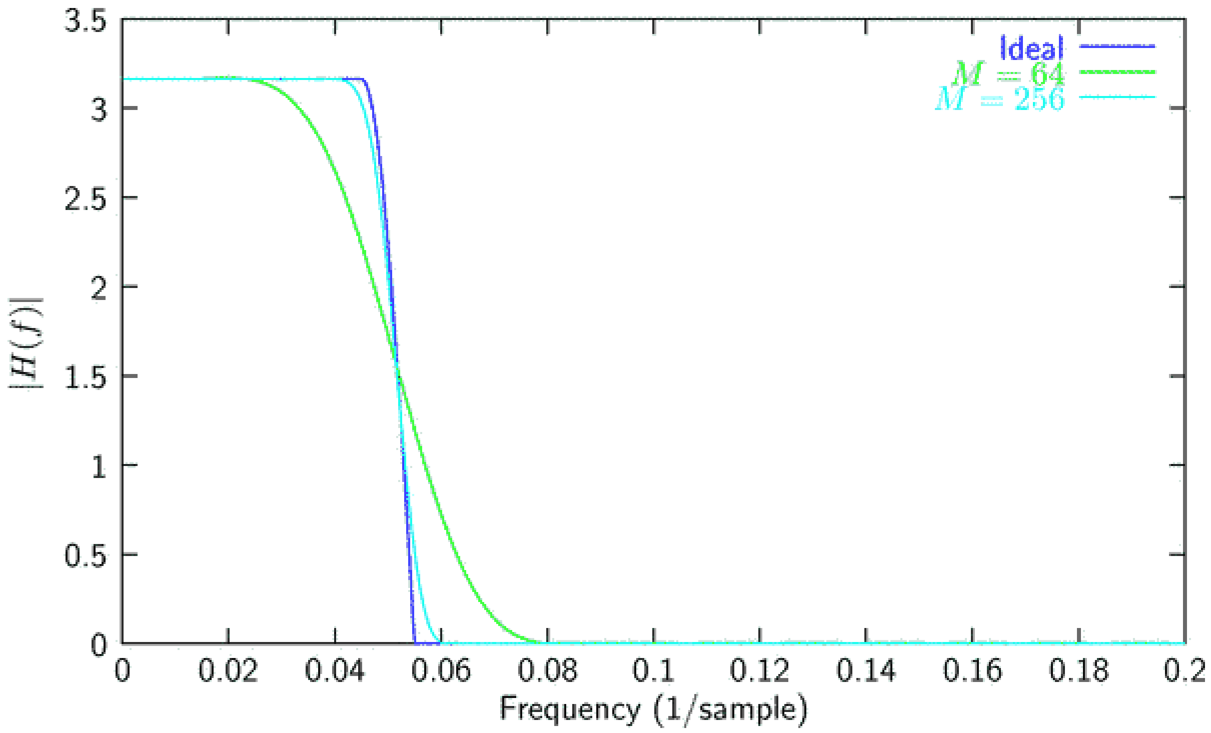
The plot above shows the response for a rectangular window. We could use this response directly, but rectangular windows tend to have high side lobes, meaning frequencies in the stop band may not be attenuated as much as we want. To avoid this, we can multiply the response by the Hamming window shown below:



resulting in the windowed time response.



To see how much deviation has occurred from our specified response, we can take the DTFT of this and compare with the ideal response, as shown below.



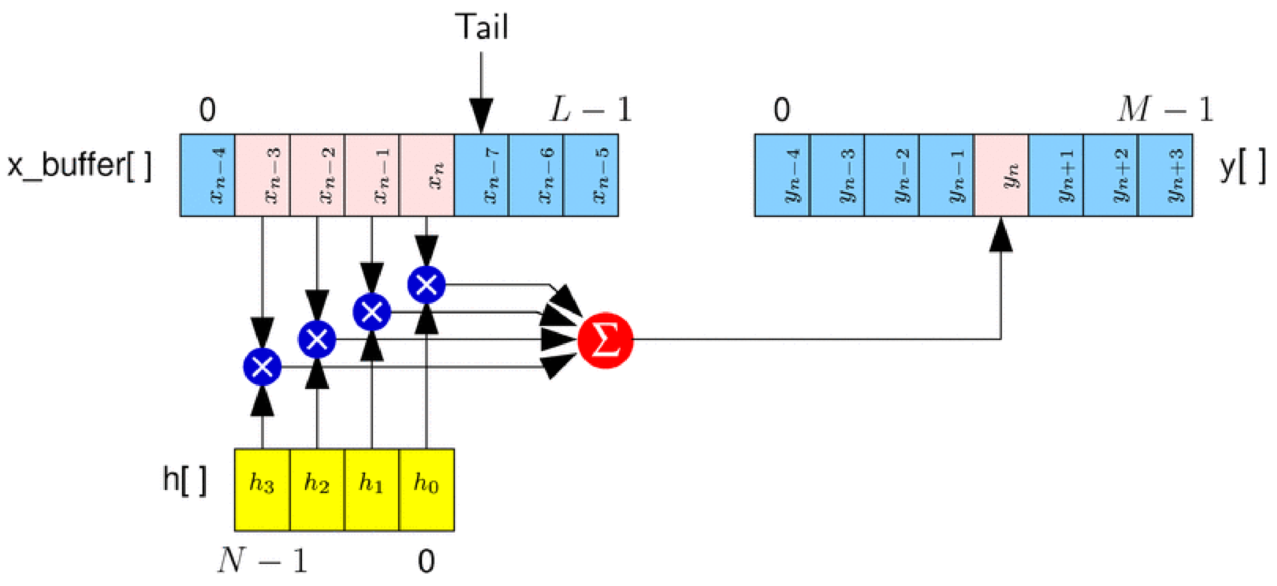
Note that the type of filter and rolloff will generally determine how many taps are needed and whether the window operation is necessary to obtain a suitable filter response.

Digital Filter Implementation

We will implement an FIR filter with the following form

$$
\begin{aligned}
y_n &= \sum_{m=0}^{N-1} h_m x_{n-m} \\
&= h_0 x_n + h_1 x_{n-1} + h_2 x_{n-2} + \dots
\end{aligned}
$$

A block diagram of a FIR filter employing a circular buffer is depicted below:



When coding this as a buffer-oriented DSP block, the block needs accomplish the following tasks:

1. A buffer of *N* samples is received.
2. Samples are moved into the circular buffer
3. The samples are filtered to generate *N* output samples
4. The *N* samples are copied into the output buffer

The required operations can be implemented using the series of steps shown in the following pseudocode.

```
for n=0:M-1,  (note M is size of input/output buffers)



   1. Move the nth sample into the circular buffer (at tail)

   2. Increment tail

   3. ptr = tail-1
```

```
        sum = 0.0

     4. for l=0:N-1,

        sum = sum + x_buffer[ptr]*h[l]

        ptr = ptr - 1

     5. Move sum into the output position

        y[n] = sum;



  end
```

Note that any increment/decrement operations on ptr and tail must be done in a circular fashion so they wrap across the boundary! As explained before, a circular increment can be done using an operation like `tail = (tail+1) AND (L-1)`. Decrement can be done with the operation`ptr = (ptr + L-1) AND (L-1)`. *Do you see why we don't just use ptr-1 here?*

Laboratory Assignment

In the lab, you will gain experience simulating a buffer-oriented DSP algorithm by creating two different blocks that store their state and employ circular buffers:

1. A simple delay block
2. A RC FIR block

First, I will walk you through the operation of the delay block where most of the details are given. Pay attention, though, since you will have to code the FIR block mostly on your own.

Delay Block

Each DSP block that we implement in MATLAB will consist of two functions:

1. [name]_init() — A function used to store parameters and initialize the block.
2. [name]() — A function that processes a buffer of input items to generate a buffer of outputs.

I recommend in this lab that you follow the naming convention above to avoid confusion.

For the delay block, the initialization function could be written as follows:

```
function [state] = delay_init(Nmax, N);
```

```
% [state] = delay_init(Nmax, N);

%

% Initializes a delay block.

%

% Inputs:

%       Nmax    Maximum delay supported by this block.

%       N       Initial delay

% Outputs:

%       state   State of block

% Notes:

%       For this block to operate correctly,

%       you should not pass in more than Nmax

%       samples at a time.


%% 1. Save parameters

state.Nmax = Nmax;


% Store initial desired delay.

state.N = N;


%% 2. Create state variables


% Make the size of the buffer at least twice of the maximum delay.

% Allows us to copy in and then read out in just two steps.

state.M = 2^(ceil(log2(Nmax))+1);


% Get mask allowing us to wrap index easily
```

```
state.Mmask = state.M-1;



% Temporary storage for circular buffer

state.buff = zeros(state.M, 1);



% Set initial head and tail of buffer

state.n_h =

state.n_t =
```

I also recommend always putting a comment block at the beginning of your functions. This imposes some structure on what you are doing and you can always type “help [func]” to remind you what the parameters are.

As you can see, the function takes parameters `Nmax`, which is the maximum delay that the block should support and `N` which is the initial delay that should be used. The function creates a *structure* called `state` that will be used to store parameters as well as the state of the block.

Next, the function creates a circular buffer for storing samples. The buffer is made an integer power of 2 in size, and a mask is computed that allows us to do the simple AND operation to ensure modulo `Nmax` indexing. The head and the tail indices are then set. *Note: for this block to work, you must properly set the initial state of these indices. Can you decide what they should be for a delay block?*

We now need to define the processing function `delay()` used to process a single buffer of samples. It is defined below:

```
function [state_out, y] = delay(state_in, x);



% [state_out, y] = delay(state_in, x);

%

% Delays a signal by the specified number of samples.

%

% Inputs:

%      state_in       Input state

%      x              Input buffer of samples
% Outputs:

%       state_out      Output state

%       y              Output buffer of samples



% Get input state

s = state_in;



% Copy in samples at tail

for ii=0:length(x)-1,

  % Store a sample

  s.buff(s.n_t+1) = x(ii+1);

  % Increment head index (circular)

  s.n_t = bitand(s.n_t+1, s.Mmask);

end



% Get samples out from head

y = zeros(size(x));

for ii=0:length(y)-1,

  % Get a sample

  y(ii+1) = s.buff(s.n_h+1);

  % Increment tail index

  s.n_h = bitand(s.n_h+1, s.Mmask);

end



% Output the updated state

state_out = s;
```

The function is very simple:

1. The input state is stored in variable `s`
2. *N* input samples are all copied to the tail of the circular buffer, where *N* is the length of the buffer. For each sample, the tail pointer is incremented.
3. *N* output samples are then read from the head of the buffer. For each sample the head pointer is incremented.
4. The updated state and output buffer `y` are then returned.

   Note that in a C program, we would not pass and copy the state this way, since it is very inefficient. Rather, we would just use a pointer to the state structure. Unfortunately, MATLAB has no way of accomplishing pointers or passing a variable by reference without using messy global variables. Realize that this is not the most efficient MATLAB code, but we do it to simulate what the C code is going to look like. Maybe one of you has a clever idea how to do this more efficiently, but still in a simple way! Please let me know if you do!

   Also note that although MATLAB uses 1-based indexing, I simulate 0-based indexing in the code. Again, this is so that when we transition to C, we do not make mistakes. My convention is to do all my indices with 0-based indexing, and then just always add 1 when I actually index the array.

   The delay block can be tested with a script like the following one:

```
% test_delay1.m

%

% Script to test the delay block.  Set up to model the way

% samples would be processed in a DSP program.



% Global parameters

Nb = 10;       % Number of buffers

Ns = 128;      % Samples in each buffer

Nmax = 200;    % Maximum delay



Nd = 10;       % Delay of block



% Initialize the delay block

state_delay1 = delay_init(Nmax, Nd);
```

```
% Generate some random samples.

x = randn(Ns*Nb, 1);

% Reshape into buffers

xb = reshape(x, Ns, Nb);



% Output samples

yb = zeros(Ns, Nb);



% Process each buffer

for bi=1:Nb,

  [state_delay1 yb(:,bi)] = delay(state_delay1, xb(:,bi));

end



% Convert individual buffers back into a contiguous signal.

y = reshape(yb, Ns*Nb, 1);



% Check if it worked right

n = [0:length(x)-1];



figure(1);

plot(n, x, n, y);



figure(2);

plot(n+Nd, x, n, y, 'x');



% Do a check and give a warning if it is not right.  Skip first buffer
in check
```

```
% to avoid initial conditions.

n_chk = 1+[Ns:(Nb-1)*Ns-1];

if any(x(n_chk - Nd) ~= y(n_chk)),

  warning('A mismatch was encountered.');

end
```

Read through the code and make sure you understand it. Comments have been provided to help you. Notice how we simulate buffer-oriented processing by partitioning the vector `x` into multiple buffers, each having 128 samples. After these buffers have been processed separately, they are then concatenated again to make the continuous buffer `y`.

At the end of the function, I generate two plots, one showing the input and output vectors, and another where the signals are plotted with the shift “undone,” meaning that the samples should line up. Finally, I do a check and issue a message if the delay did not work correctly. Maybe all this checking seems like overkill, but it is important to check your blocks in isolation first before you hook things together. This incremental design strategy is the only way to avoid extreme frustration later!

Do the following with the delay block:

1. Test it with a few different delays and convince yourself it works.
2. Try cascading two delay blocks. Realize this means you have to call the `delay_init()` function and use two different state variables. Make sure it works!

FIR Filter

Next, I want you to implement the FIR filter using the same basic idea. You will create two functions. I will give you the skeleton for these functions below, which you need to fill out with code:

```
function [state] = fir_init(h, Ns);



% [state] = fir_init(h, Ns);

%

% Creates a new FIR filter.

%

% Inputs:

%      h              Filter taps
%       Ns              Number of samples processed per block

% Outputs:

%       state           Initial state



%% 1. Save parameters



%% 2. Create state variables



% Make buffer big enough to hold Ns+Nh coefficients.  Make it an
integer power

% of 2 so we can do simple circular indexing.



% Temporary storage for circular buffer



% Set initial tail pointer and temp pointer (see pseudocode)

state.n_t =

state.n_p =

function [state_out, y] = fir(state_in, x);



% [state_out, y] = fir(state_in, x);

%

% Executes the FIR block.

%

% Inputs:

%       state_in        Input state

%       x               Samples to process

% Outputs:
```

```
%       state_out      Output state

%       y              Processed samples


% Get state

s = state_in;


% Move samples into tail of buffer


% Filter samples and move into output


% Return updated state

state_out = s;
```

You could use the following code to check your filter:

```
% test_fir1.m

%

% Script to test the FIR filter.


% Global parameters

Nb = 100;      % Number of buffers

Ns = 128;      % Samples in each buffer


% Generate filter coefficients

p.beta = 0.5;

p.fs = 0.1;

p.root = 0;    % 0=rc 1=root rc

M = 64;
```

```
[h f H Hi] = win_method('rc_filt', p, 0.2, 1, M, 0);


% Generate some random samples.

x = randn(Ns*Nb, 1);


% Type of simulation

stype = 0;     % Do simple convolution

%stype = 1;    % DSP-like filter


if stype==0,

  y = conv(x, h);

elseif stype==1,

  % Simulate realistic DSP filter


  % ADD YOUR CODE HERE !!!


else

  error('Invalid simulation type.');

end


% Compute approximate transfer function using PSD

Npsd = 200;    % Blocksize (# of freq) for PSD

[Y1 f1] = psd1(y, Npsd);

[X1 f1] = psd1(x, Npsd);

plot(f1, abs(sqrt(Y1./X1)), f, abs(H));

xlim([0 0.2]);
```

The function sets some global parameters (number of buffers and size of each buffer), and then creates the filter coefficients for a raised cosine filter with the parameters shown. The `stype` variable controls whether a direct convolution is done (like in a normal MATLAB program) or a detailed real-time DSP simulation is done. In this second case, you need to add your code, similar to what was done in the test_delay1.m!

Do the following:

1. Run the `test_fir1.m` code for just the `stype=0` case. You should see that the H(f) looks very similar to the specified response in the output plot.
2. Implement your FIR block completely and add the necessary code to the test script. Make sure you can get the same response as the ideal case above. If it doesn't work, I find it helpful to put breakpoints ("keyboard" statements) in the code to allow me to check and plot key variables. You should be able to track down the problem this way.

Check-Off

Demonstrate the following things to the TA:

1. Explain the difference between non-real-time MATLAB processing and real-time DSP code.
2. Demonstrate that the delay block works correctly. Also show how you set the initial head and tail pointers and how this gives the delay you want.
3. Show that cascaded delay blocks work.
4. Demonstrate your FIR filter and show that you get close to the specified H(f).

Lab Write Up

For Delay lab, provide the following items in the write up:

1. A short explanation of the difference between a usual MATLAB simulation and real-time DSP code. Also, briefly explain why buffer-oriented processing is needed and what affects the latency and throughput of an algorithm.
2. A diagram showing conceptually how your Delay uses a circular buffer to implement the Delay.
3. A printout of the MATLAB code that implements your Delay with comments.

For FIR lab, provide the following items in the write up:

1. A diagram showing conceptually how your FIR filter uses a circular buffer to implement the FIR equation.
2. A printout of the MATLAB code that implements your FIR filter with comments.
3. Plots showing the ideal response of your filter compared to the simulated response of the filter. Explain any discrepancies.
4. Any problems you ran into in the lab and how you fixed them.